Distributed Chat Server
Milestone - 2

Group Members :
Nicholas Fausti (nfausti)
Nikhilesh Behera(nbehera)
Samarth Shah(ssamarth)

Distributed Chat Server implemented in C language

Constraints for the Distributed Chat server -

1. The messages that are exchanged can have maximum length of 220 characters
2. There can be a maximum of 20 users in the chat group
3. Length of the username can have a maximum of 14 characters
4. 2 users cannot have the same username
5. All usernames and messages sent are case-sensitive

Data Structures -

1. Table (Array of structures) contains the list of users that are present in the chat group and also contains details of each users. Table contents are as follows :
    a. ID number : ID # associated with that user which will be used during leader election
    b. IP address : Address for that given user
    c. Port number : Port number corresponding to that user
    d. isLeader : Column to keep track of which user is the leader at any given point of time
    e. isActive : To check whether any given user is active or not
    f. timerBroadcastCheck : A timer is maintained for each user and keeps track whether or not the table has been received after broadcast
    g. timerMsgBroadcastCheck : A timer is maintained for each user and keeps track whether or not the messages have been received after broadcast
    h. receivedMsgSeqNo : Sequence number for every given user that keeps track of any duplicacy while sending to the leader
    i. updatedNewEntryTimeStamper : This sequencer value at the leader side is broadcasted at every time a new user joins the chat group, hence keeps the value updated at all users.
2. Send Queue : A queue that contains all the messages that are typed in the stdin and are sent to the leader periodically

3. Receive Queue : A queue that contains all the incoming messages from the leader after broadcast
4. Send Backup Queue : A queue that contains a copy of the messages that are sent to the leader in order to accommodate lost messages.
5. Holdback Insertion List : A list that is maintained to store out of order messages that are received by the user at any given point of time. This data structure also sorts the incoming data based on the sequence number using insertion sort.
6. Global Send Queue : A queue at the leader side that captures all the messages to be broadcasted to the users. This queue contains the messages after they are assigned a sequence number.
7. Available ID Queue : A queue containing all the ID numbers that can be reused due to user crash/exit.

Message Types -

1. “Add~<Username>” : Message sent during connection request/request to join the group
2. Table broadcasted to all the users in case of add request successful
3. “AckTable” : Acknowledgement sent by all the users on receipt of the table from the leader
4. “Exists~<Message>~<Username>~<Seq1#>” : When any user sends a message to the group hence sends it to the leader
5. “Message~<Username:: (message)>:Seq#” : Message broadcasted to all the users with the global sequence number value
6. “AckRecvdBroadcastMsg” : Acknowledgement sent by all the users on receipt of the broadcasted message from the leader
7. “Error-1” : Leader sends out the message when there are 20 active users in the group
8. “Error-2” : Leader sends out this message when the given username (case-sensitive) already exists in the group
9. “BecomeLeader~<Username>” : Incase of leader crash detected then the user who detected the crash sends out the message to the next leader
10. “String” or “Table” : While the leader sends messages to the users in the group it first sends out “String”/”Table” depending on the data type it wants to send

Design Implementation -

Execution flow :

Leader : Initiates the chat group
User :

- Connects to the group(made by the leader) by using the ip address and the port number of the leader
- Sends a request message: “Add~(username)” to the leader Incase a user contacts a non-leader to be added to the group, this non-leader sends out the information(ip address and the port number) of the current leader to the user requesting so that it can contact the current leader to be added to the group chat

Leader :

- Receives the string from the user, parses it to get a Add request and updates the table by adding the user details of the new user in the table.
- Sends the updated table to all the current active users in the chat group so that all the users have a new entry created for the new user.
- This also plays the role of an acknowledgement for the user. If the new user receives the table, it need not resend the add request to the leader. In case it does not receive the table after timeout and 3 successive tries, it concludes leader has crashed.

Chatting system :

Throughout the chat, the communication between the leader and the user takes place via 2 way handshake. Any user at any particular point of time can accept two kinds of messages.

1. String Identifier(Table or String) to classify that the succeeding message it would be receiving will be of a table (database of the users) or a text message meant for the group or the acknowledgement message meant for the user.
2. A table or a string message depending on the identifier it received before.

- Each member of the group chat sends the message it wants to send to the group with a string identifier “Exists” pre-attached to the message, first to the leader.
- The user enqueues this message in the sendQ first.
- A thread constantly runs to dequeue this sendQ when it is filled with messages and sends the message to the leader with a pre attachment Exists.
- The leader then parses the string, and depending on what identifier it receives executes the succeeding tasks.
- If it receives “Exists” as the first parsed string, then it carries the following actions:

- It checks for the sequence number of the message , appended with the message it received from the user to check for duplicacy. Ignores duplicates messages. If the sequence number is one more than the previous one, then the leader accepts the message, timestamps it and broadcasts it to all the users.
- All the members of the group then receives this broadcasted message, enqueues it in a holdbackQ and checks for the time stamp associated with every message. If the timestamp is the next expected one, then the message dequeus from the holdBackQ and enqueues it in a recvQ. A thread is running to constantly dequeue the recv queue and print the messages out on the console of the user. Note that all the messages enqueued in the recvQ will be in the right order. of timestamp and thus ensures total ordering.
- The holdbackQList is constructed in the form of a list basically, in which there is constant sequencing going on depending on the timestamp and at any particular point of time, the holdbackQList always has the next highest timestamped message as its head which can be readily removed from the holdbackQList. This operation constantly happens inside a separate thread.
- At every operation that is taking place, there are acknowledgements for every message transfer that is taking place.
- User on receipt of a table sends an acknowledgement AckTable to the leader.
- Leader receives this acknowledgement and goes ahead. If the leader does not receive this acknowledgement from a particular user, for 3 successive tries for a particular timeout, then it resends the table to that particular user.
- Similar is the case for the broadcast message sent out by the leade to all the users. In case, leader does not receive an acknowledgement from all users, it resends the message out to that particular user after 3 tries and a timeout.
- Note: Duplicacy is taken care of by a check by the leader that which user has sent out the acknowledgement and who has not. User resends the table/broadcast message only to those particular users from whom it has not received the acknowledgement.

Leader Crash/Exit Detection :

- Leader crash is detected lazily. When a user wants to send a message to the group, it first sends it to the leader. The leader on receipt of this message waits for an acknowledgement(Broadcasted message itself) from the leader. Incase it does not receives this message for three consecutive tries after a timeout, it declares that the leader has died/crashed.
- In such a case, it conducts a leader election. The details about the leader election is explained in next section below.
- In case a leader exits willfully, and leaves the group, it first informs the next user with the next greatest ID asking him to become the leader. When it receives an acknowledgement from the new leader, then it leaves the group, else it resends the request to the succeeding greatest ID user and so on.

User Crash/Exit Detection :

- In case the user crashes, we detect this in two ways:
    - Lazily:
      When the leader receives a message from a user, which it then broadcasts to all the users, it waits for an acknowledgement from every user about the receipt of the message. In case it does not receive, the ack after 3 tries, leader concludes that the user has crashed. In this way, we can keep a track of all the active users
    - Heartbeat:
      Leader periodically pings every user of the group (heartbeat). The user on receiving the ping responds by sending “Alive”. In this way, the leader dynamically maintains the list of all the active users
- There is a clean up thread that constantly runs in the background checking for the active users, and in case a leader concludes that a user has crashed, it removes the user from the table, and broadcasts the updated table with all the active users periodically.

Leader Election Algorithm -

The leader election algorithm that we have implemented is similar to the bully algorithm but there are few minute difference and also the time complexity of our algorithm is O(n). In our algorithm as we have limited number of users in the group (20), we assign a ID number to all the users and based on their entry into the group. The first user to enter the group becomes the leader with an ID number of 20 and subsequent users entering the group have ID number decremented by 1. In this way we always pick the highest ID number to become the leader in case of leader crash/exit. Also to keep track of the users that are active in the group we have a column in our global table that keeps track of the same. In this way at all times we have the highest active ID number to become the leader. The trade off in this algorithm would be that we have increased space complexity and reduced time complexity. As for a chat server time is more essential.

UDP reliable and robust -

As we know that UDP protocol seems to be very unreliable. To make the protocol reliable and robust we have implemented certains methods that would take care of message loss, message duplicacy and also out of order delivery of messages.

1. Handling Out-Of-Order messages : To handle the case where users might receive messages in different order we have implemented a total ordering of messages where the leader acts as a global sequencer and appends a sequence number to all the outgoing messages. Also on receipt of any messages which are not in order stored in a holdback insertion list. This list contains all the messages that are out of order and also in a sorted order and are delivered onto the stdout when it matches with the local sequence number.
2. Handling message loss : To handle lost messages we have used acknowledgements to detect if the message was delivered or not. If the ACK is received then the message was delivered successfully or else the message was lost. On losing messages we have implemented a timeout which helps in retransmission of the lost message. Upon 3 retries if the message still can't be delivered then the recipient of the message is declared dead.
3. Handling message duplicacy : The global table consists of a column that contains a sequence number corresponding to the number of messages sent by any given user. Hence on receiving a message the leader checks for the sequence number in the table for that given user and decides whether it is a new message or an old duplicate message and accordingly accepts/drops the message.

Logging -

In order to maintain the codebase and also for debugging purposes we have decided to log all the messages that are being sent and received to by any given user. In this way it allows us for thorough debugging of all the states that can occur.

Distribution of work:
Communication between user and leader: Nikhilesh
Implementation of chat system: Samarth
Acknowledgements/Reliable: Nicholas
Leader election: Nikhilesh
Total ordering: Samarth
Duplicacy: Nicholas
Logging: Nikhilesh
Two way handshake: Samarth
Modularisation: Nicholas
Leader crash detection: Nikhilesh
User crash detection: Samarth
Code clean-up: Nicholas, Nikhilesh, Samarth
Commenting: Samarth
Performance testing- Nicholas

Extra credit:
Traffic control: Nikhilesh, Samarth, Nicholas
Fair queueing: Samarth
Message priority: Nikhilesh
Encrypted chat messages: Nicholas
Advanced election criteria: Nikhilesh, Nicholas, Samarth
Decentraized total ordering: Nikhilesh, Samarth, Nicholas

Extra Extra credit: :D
Web based application: Nicholas